VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

REIMPLEMENTACE NÁSTROJE COMBINE

A REIMPLEMENTATION OF THE COMBINE TOOL

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **MATÚŠ VRÁBLIK**
AUTHOR

VEDOUCÍ PRÁCE **Ing. ALEŠ SMRČKA, Ph.D.**
SUPERVISOR

BRNO 2024

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání bakalářské práce

155684

Ústav:	Ústav inteligentních systémů (UITS)
Student:	**Vráblik Matúš**
Program:	Informační technologie
Název:	**Reimplementace nástroje Combine**
Kategorie:	Analýza a testování softwaru
Akademický rok:	2023/24

Zadání:

1. Nastudujte metodu kombinačního testování. Nastudujte algoritmus IPOG, jeho rozšíření a prototypovou implementaci v nástroji Combine vyvíjeném na FIT VUT.
2. Analyzujte slabiny současné implementace. Navrhněte reimplementaci nástoje Combine.
3. Reimplementujte nástroj Combine. Klaďte důraz na spolehlivost a udržitelnost zdrojových kódů.
4. Vyhodnoťte novou verzi nástroje Combine.

Literatura:

- IEEE/ISO/IEC 29119-4-2021 International Standard - Software and systems engineering--Software testing--Part 4: Test techniques.
- Kacker, R. (2013), An Efficient Algorithm for Constraint Handling in Combinatorial Test Generation, Proceedings of Sixth IEEE International Conference on Software Testing, Verification and Validation ICST 2013, Luxembourg, -1, [online], https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=913191
- Nástroj Combine, https://combine.testos.org/ . Zdrojové kódy dostupné na: https://pajda.fit.vutbr.cz/testos/combine-bcc

Při obhajobě semestrální části projektu je požadováno:
První dva body zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce:	**Smrčka Aleš, Ing., Ph.D.**
Vedoucí ústavu:	Hanáček Petr, doc. Dr. Ing.
Datum zadání:	1.11.2023
Termín pro odevzdání:	9.5.2024
Datum schválení:	6.11.2023

Abstrakt

Cílem bakalářské práce je pochopení kombinačního testování, algoritmu IPOG na tvorbu kombinačních testů, jeho rozšíření o obmedzení hodnot parametrů a prototypovou implementaci v nástroji Combine vyvíjeném na FIT VUT. Následne analyzovat slabiny současné implementace, navrhnout reimplemetaci a také nástroj reimplementovat, porovnat s původní verzí a vyhodnotit novou verzi nástroje Combine.

Abstract

This bachelor thesis aim is understanding of combinatorial testing, algorithm IPOG used to create combinatorial tests, its extension by limiting parameter values and prototype implementation in tool Combine developed at FIT VUT. Then analyze weaknesses of current implementation, propose reimplementation, also reimplement the tool, compare with original version and evaluate new version of Combine tool.

Klíčová slova

IPOG, IPOG-C, kombinační testy, combine, combineng, CLI applikace

Keywords

IPOG, IPOG-C, combination tests, combine, combineng, CLI application

Citace

VRÁBLIK, Matúš. *Reimplementace nástroje Combine*. Brno, 2024. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Aleš Smrčka, Ph.D.

Reimplementace nástroje Combine

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Aleše Smrčky, Ph. D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Matúš Vráblik
7. května 2024

Poděkování

V sekci poděkování bych rád vyjádřil upřímnou vděčnost svému skvělému školiteli, Ing. Aleši Smrčkovi, Ph.D., za jeho cenné rady, odborné vedení a trpělivost během celého průběhu mé bakalářské práce.

Obsah

Seznam obrázků

Kapitola 1

Úvod

Tato práce se zabývá studiem metod kombinačního testovaní. V této bakalářské práci jsme se konkrétně zaměřili na algoritmus IPOG. Tento algoritmus představuje deterministický nástroj pro tvorbu kombinačních testů nespecifický pro žádný systém jak vyplývá ze samotného názvu IPO-*generalised* tedy IPO-obecný algoritmus. Cílem IPOG algoritmu je vytvořit co nejmenší sadu testů, s pokrytím všech T-tic parametrů, za co nejkratší čas pro zadané parametry s jejich bloky hodnot. Pro smysluplný výsledek je žádoucí použít omezení na hodnoty parametrů. Například mějme parametry „ve vzduchu“ a „na souši“, bez omezení by mohl vzniknout nesmyslný test kde budou oba parametry mít hodnotu "True". Omezení řeší tento problém. Tomuto se věnuje rozšířeni IPOG-C, které přidává omezení na hodnoty parametrů. V této bakalářské práci se následně analyzují slabiny původního prototypového nástroje Combine, který implementuje IPOG-C v jazyku Python. Na základe analýzy původního nástroje byli vytvořeny návrhy zlepšení tohoto nástroje jako je implementace algoritmu IPOG-C v jazyku C++, vhodnější datové struktury pro sekvenční průchod, využití multithreadingu, atd. Výsledkem této bakalářské práce je funkční reimplementace se zachováním původní funkcionality. Práce porovnává novou a původní verzi nástroje respektive Combine a CombineNG. Závěrem je úspěšné zrealizování reimplementace, která je rychlejší než původní, no není zcela deterministická a má větší paměťové nároky.

Kapitola 2

Testovací metody

Pro zlepšení kvality softwaru bylo navrženo a studováno mnoho metod jeho testování. Různé testovací metody se zaměřují na různé typy chyb a mohou být účinné pro určité testovací scénáře. Například testování mutací, které zahrnuje modifikaci zdrojového kódu v malém měřítku, může pomoci testerovi vyvinout účinné testy, identifikovat slabiny v testovací sadě nebo najít části kódu, které jsou použité zřídka nebo nikdy během vykonávání. Metamorfní testování může vyřešit problém orákula při testování na základě vlastnosti Software Under Test (SUT). Strukturální testování má za cíl najít chyby související s vnitřní strukturou SUT. V této bakalářské práci se zaměřujeme na kombinatorické testování (CT), také tzv. Kombinatorické testování interakcí (CIT). CT testuje SUT s testovací sadou krycího pole který testuje všechny požadované kombinace hodnot parametrů. Výhodou CT je že dokáže detekovat poruchy vyvolané interakcemi mezi parametry v SUT[5].

V testování softwaru neexistuje žádná směrnice, která by diktovala nejlepší způsob testování, a neexistují žádné „nejlepší postupy“, které by vždy mohly zaručit úspěch. Ačkoli je CT užitečné při zjišťování určitých chyb, může vytvářet falešnou důvěru, protože[5]:

1. CT může být vnímáno jako způsob, jak poskytnout určitou zkratku testování softwaru. Má to svá úskalí, jako je netestování všech možných kombinací parametrů.

2. Pokud parametry a jejich hodnoty nejsou správně zvoleny, sníží se tím schopnost CT detekce defektů.

3. Pokud se nám nepodaří identifikovat všechny interakce mezi parametry v SUT, CT tyto „chybějící“ interakce nebude testovat.

4. Pokud nemáme „dost dobrý“ orákulum, ověření výsledků testování bude obtížné.

Na generování kombinačních testů existují víceré algoritmy avšak mnohé jsou jenom pro specifické SUT. Proto se táto práce soustředí na všeobecný algoritmus In-Parameter-Order-General (IPOG) algoritmus.

2.1 Algoritmus IPOG

Tento algoritmus byl vytvořen zobecnením IPO strategie ze dvou důvodu. První důvod, aby strategie nekladla omezení na testovanou systémovou konfiguraci(SUT - *system under test*). Kvůli tomu je vhodnější výpočetní přístup oproti algebraickému přístupu. Druhý důvod, obecné t-násobné testování má přísnější časové a paměťové nároky než párové testování.

```
Algorithm IPOG-Test (int t, ParameterSet ps)
{
1. initialize test set ts to be an empty set
2. denote the parameters in ps, in an arbitrary order, as P1, P2, ..., and Pn
3. add into test set ts a test for each combination of values of the first t parameters
4. for (int i = t + 1; i ≤ n; i ++){
5.    let π be the set of t-way combinations of values involving parameter Pi
        and t -1 parameters among the first i – 1 parameters
6.    // horizontal extension for parameter Pi
7.    for (each test τ = (v1, v2, ..., vi-1) in test set ts) {
8.       choose a value vi of Pi and replace τ with τ' = (v1, v2, ..., vi-1, vi) so that τ' covers the
           most number of combinations of values in π
9.       remove from π the combinations of values covered by τ'
10.   }
11.   // vertical extension for parameter Pi
12.   for (each combination σ in set π){
13.     if (there exists a test that already covers σ) {
14.        remove σ from π
15.     } else {
16.        change an existing test, if possible, or otherwise add a new test
             to cover σ and remove it from π
17.     }
18.   }
19.}
20.return ts;
}
```

Obrázek 2.1: Algoritmus IPOG-test[4]

Z toho důvodu je vhodnější strategie IPO než ostatní jako například AETG(*Automatic Efficient Test Generator*) a techniky heuristického vyhledávání. Také poznamenáme, že IPO strategie je deterministická. Struktura IPOG strategie může být popsána následovně: Pro systém s t nebo více parametry, IPOG strategie vytváří t-násobnou testovací sadu pro prvních t parametrů, rozšiřuje testovací sadu o vytvoření testovací sady t-násobní pro první $t+1$ a poté pokračuje v rozšiřování testovací sady dokud nevytvoří t-násobní testovací sadu pro všechny parametry. (Parametry mohou být v libovolném pořadí.) Rozšíření existující t-násobní testovací sady pro další parametr je provedeno ve dvou krocích:

- horizontální růst, který rozšiřuje každý existující test přidáním jedné hodnoty pro nový parametr
- vertikální růst, který v případě potřeby přidává nové testy, na testovací sadu vytvořenou horizontálním růstem

Obr. 2.1 ukazuje algoritmus generování testu nazývaný IPOG-Test který implementuje tuto strukturu. Algoritmus bere jako vstup celé číslo t a sadu ps parametrů a vytvoří jako výstup t-násobnou testovací sadu pro parametry v sadě ps. Je předpokládáno, že počet n parametrů v sadě ps je větší než nebo rovno t[4].

[Variable]	[Constraints]
a: 0, 1, 2	(1) a + b > c
b: 0, 1, 2	(2) a = 0
c: 0, 1, 2	(3) b = 0

Obrázek 2.2: Ukázka příkladu problému uspokojení omezení[6]

2.2 Algoritmus IPOG-C

Jde o rozšíření algoritmu IPOG v sekci 2.1 o zvládání omezení na bloky vstupných parametrů.

Praktické aplikace mají často omezení, jak lze hodnoty parametrů kombinovat v testu. Někdo může chtít například zajistit, aby webová aplikace mohla být správně spouštěna v různých webových prohlížečích běžících na různých operačních systémech. Vezměte v úvahu, že Internet Explorer (nebo IE) 6.0 nebo novější nelze spustit na MacOS. Pokud je tedy webový prohlížeč IE 6.0 nebo novější, operační systém nesmí být MacOS. Toto omezení je třeba vzít v úvahu, aby se IE 6.0 nebo novější a Mac OS neobjevily v stejném testu[6].

Omezení (constraints) musí být definovány před jejich použitím v generování testů. Mohou být ve tvaru zakázaných n-tic, ale může být problematické je vyjmenovat. Použití množiny logických výrazů řeší tento problém a přináší stručnější zadání omezení. Algoritmus IPOG-C se snaží vyhnout volání do funkce řešení omezení. Když se volání vyhnout nedá, tak se snaží zjednodušit proces řešení co nejvíce. Proto obsahuje následující tři optimalizace[6]:

- *Vyhnutí se zbytečným kontrolám platnosti na t-násobných kombinacích.*
- *Kontrolování jenom příslušných omezení.*
- *Zaznamenávání historie řešení.*

Na obr.2.3 jsou zvýrazněny změny oproti původnímu IPOG algoritmu. Původní algoritmus je zachován a obohacen o zpracování omezení, takže v případě tvorby sady testů bez omezení dostaneme stejné výsledky jako u původního. Při kontrole validity testu se pošle samotné omezení a potřebné parametry z testu.

Uvažujme, že systém se skládá ze 3 parametrů a, b, c, z nichž každý má 3 hodnoty a jedno omezení „a + b > c“. Obr. 2.2 ukazuje problém uspokojení omezení pro kontrolu platnosti kombinace a.0, b.0. Všimněte si, že dvě omezení „a = 0“ “ a „b = 0“ jsou přidána pro hodnoty parametrů a.0 a b.0, navíc k omezením zadaným uživatelem, tj. „a + b > c“.[6]

Optimalizace algoritmu IPOG-C

Následující optimalizace algoritmu byli převzaty z [6].

Zamezení nepotřebným kontrolám validity cílových kombinací

V 2.3 se na 3. řádku vygenerují platné testy pro prvých t parametrů. Z toho plyne že všechny testy v setu *ts* jsou platné. Další důležité pozorování je fakt, že mezi kontrolou platné kombinace v 2.3 na 5. řádku a platného nového testu na 8. řádku je značné překrytí

```
Algorithm IPOG-C (int t, ParameterSet ps)
{
1. initialize test set ts to be an empty set
2. sort the parameters in set ps in a non-increasing order of their
      domain sizes, and denote them as P1, P2, ..., and Pk
3. add into test set ts a test for each valid combination of values
      of the first t parameters
4. for (int i = t + 1; i ≤ k; i ++){
5.    let π be the set of all valid t-way combinations of values
         involving parameter Pi and any group of (t-1) parameters
         among the first i-1 parameters
6.    // horizontal growth for parameter Pi
7.    for (each test τ = (v1, v2, ..., vi-1) in test set ts) {
8.       choose a value vi of Pi and replace τ with τ' = (v1, v2, ...,
            vi-1, vi) so that τ' is valid and it covers the most
            number of combinations of values in π
9.       remove from π the combinations of values covered by τ'
10.   } // end for at line 7
11.   // vertical growth for parameter Pi
12.   for (each combination σ in set π){
13.      if (there exists a test τ in test set ts that can be changed to
            a valid test τ' that covers both τ and σ {
14.         change test τ to τ'
15.      } else {
16.         add a new test only contains σ to cover σ
17.      } // end if at line 13
18.   } // end for at line 12
19. }// end for at line 4
20. return ts;
}
```

Obrázek 2.3: Algoritmus IPOG-C

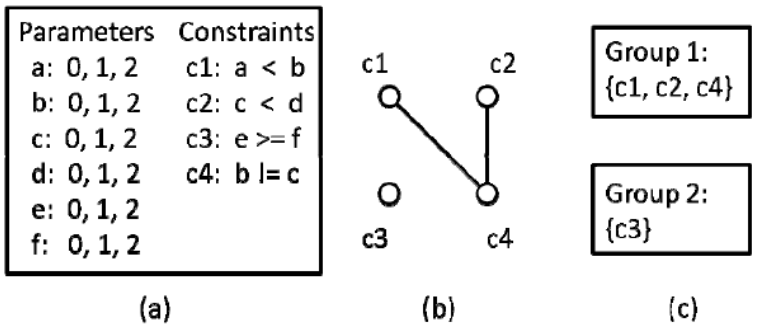
co značí zbytečné kontroly. Kontrolování celého setu kombinací, který je obvykle velký, je náročné a také zbytečné.

Proto navrhujeme následnou optimalizaci. Přesunout kontrolu platnosti kombinací v 2.3 na 5. řádku přesunout na 12. řádek , který by vypadal takhle „**for**(each **valid** combination σ in set π)“. Tímto způsobem se každá kombinace v setu zkontroluje prvý krát na 8. řádku, mnohem menší počet kombinací druhý krát na 12. řádku, co značně redukuje počet kontrol platnosti.

Kontrolování jenom příslušných omezení

Za předpokladu sada testů *ts* obsahuje jenom platné testy, není zapotřebí kontrolovat platnost všech omezení. Proto tahle optimalizace navrhuje kontrolu jenom příslušných omezení, co zjednoduší daný problém splněni omezení.

Prvně se omezení rozdělí do skupin omezení společných dle parametrů. Omezení která sdílí parametr budou v jedné skupině. Z obrázku 2.4 vidíme omezení *c1* a *c2* jsou v jedné



Obrázek 2.4: Zobrazení skupin omezení[6]

skupině kvůli parametrům *b* a *c* , které se nachází v omezení *c4*. Takto se omezení dostanou do skupin na základě společných parametrů přímo nebo nepřímo.

Protože algoritmus IPOG generuje testy postupně, tato optimalizace může být velmi efektivní. Pro pokrytí nového parametru *p* stačí zkontrolovat skupinu omezení do které parametr *p* patří.

Zaznamenávání historie řešení

V této optimalizaci uložíme historii řešení pro každou skupinu omezení abychom se vyhnuli řešení jednoho problému víckrát. Ukládáme parametry pro které byly omezení vyřešeny a výsledek řešení. Při opětovné žádosti o řešení se nejdřív zkontroluje historie. Pokud řešení obsahuje, vrátí se uložená hodnota. V opačném případe se omezení vyřeší a výsledek se uloží do historie příslušné skupiny.

Kapitola 3

Současná (prototypová) implementace

Následující 2 paragrafy byli převzaty z [2]. Combine je webový nástroj umožňující specifikovat vstupní parametry SUT a vygenerovat T-Wise *covering array*, testovací sadu, která pokrývá všechny T-násobné kombinace bloků vstupních parametrů. Nabízí možnost specifikovat parametry 10 datových typů, sílu T pro specifikaci pokrytí výskytu jednotlivých bloků ($T = 1$) až šestic ($T = 6$) v testovací sadě.

Nástroj nabízí jak možnost generovat indexy jednotlivých bloků parametrů v jednotlivých testovacích případech, tak i možnost generovat testovací sadu s konkrétními generovanými hodnotami jednotlivých bloků. Dále je také možnost specifikovat omezení kombinací bloků a tím tak vyloučit kombinace bloků parametrů, které nedává smysl nebo nelze testovat (např. testovat webovou aplikaci v prohlížeči Microsoft Edge na operačním systému Ubuntu). Vygenerovanou testovací sadu nástroj nejen zobrazí, ale také nabídne možnost sadu stáhnout ve formátu JSON, XML a CSV.

Prototyp ale není vhodný na použití s vyšší sílou testu T u většího počtu parametrů s více bloky. Pro takovýto vstup nástroj vypočítá výsledek v rádu desítek minut až hodin. Tady se ukazují slabiny programovacího jazyku Python a to vykonávání programu na jednom vláknu, jde o jazyk vysoké úrovně abstrakce a fakt že jde o interpretovaný jazyk.

3.1 Analýza nástroje Combine

Hlavním indikátorem efektivity implementace jsou 2 faktory: doba vykonávání a velkost vytvořené testovací sady. Správnost sady tedy pokrytí T-násobných kombinací je samozřejmostí. Doba vykonávání byla měřena získáním času pomocí vestavěných časových funkcí před(t_1) a po(t_2) vypočítání testovací sady a rozdílem těchto dvou hodnot(Δt).

$$\Delta t = t_2 - t_1$$

Z pohledu správnosti testovací sady je Combine v pořádku dle testů srovnávajících vytvořenou sadu testů vůči očekávané. Počet testů v sadě je blízký počtu z nástrojů ACTS a Jenny. Combine implementuje deterministický algoritmus IPOG. Budeme tedy zkoumat způsoby krácení doby běhu bez změny výstupu. V prvním kroku jsme zjistili pomocí profileru které části kódu se vykonávají nejdéle. K tomuto jsme použili celkový čas strávený v dané části. Zkoumání se provádělo se vstupem 10 parametrů o 10 blocích a síle 3, protože při malém počtu parametrů byla doba běhu krátká a také proto že nejvíc celkové doby běhu se trávilo

```
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    68547   48.471    0.001   56.670    0.001 ipog.py:735(_check_if_covered)
    63470   35.626    0.001   40.921    0.001 ipog.py:788(_try_dont_care_values)
   124732   28.464    0.000   37.188    0.000 ipog.py:473(_get_best_value_for_test_case)
316399693/316398737   16.684    0.000   16.685    0.000 {built-in method builtins.len}
   249464   10.760    0.000   10.760    0.000 {method 'index' of 'list' objects}
   124732   10.152    0.000   16.705    0.000 ipog.py:665(_remove_combinations)
  5108341    5.909    0.000    5.909    0.000 ipog.py:521(<listcomp>)
  5108341    5.908    0.000    5.908    0.000 ipog.py:703(<listcomp>)
      372    2.288    0.006    2.591    0.007 ipog.py:358(_insert_value_combinations)
996618/21    0.871    0.000    1.618    0.077 copy.py:128(deepcopy)
```

Obrázek 3.1: Profiling nástroje Combine pomocí python modulu cProfile

v přípravných funkcích volaných málo krát. Profiling na obr. 3.1 ukázal na 5 částí které zabírali nejvíc času:

- **__check__if__covered** prochází sadu testů a porovnává každý test s kombinací kterou hledá
- **__try__dont__care__values** prochází sadu testů a hledá test s „dont care values“ tak aby je mohla změnit a pokrýt hledanou kombinaci
- **__get__best__value__for__test__case** prohledá nepokryté kombinace pro aktuální parametr a vybere blok který pokryje nejvíc z těchto kombinací
- **__remove__combinations** odstraní kombinace pokryté testem z nepokrytých kombinací
- **<listcomp>** nebo „List comprehension“ je způsob vytváření kopie seznamu.

Z tohoto krátkého popisu funkcí vidíme místa kam se v kódu podívat a kde se soustředit při optimalizaci.

Návrh zlepšení v reimplementaci

Popis funkcí `_check_if_covered` a `_try_dont_care_values` je podobný v tom že oba prochází sadou testů. Dále jsou v kódu téměř hned za sebou:

```
1071                if _check_if_covered(value_comb, ts) == 0:
1072                    # remove value_comb from pi
1073                    pi[param_comb].remove(value_comb)
1074
1075                    if pi[param_comb] == set([]):
1076                        del pi[param_comb]
1077
1078                else:
1079                    ret_val = _try_dont_care_values(value_comb, ts)
```

Musím podotknout šikovné využití proměnné `dont_care_values`, která obsahuje indexy jenom do testů obsahující alespoň jednoho žolíka, co v konečném důsledku redukuje počet kontrol na nezbytné minimum. V nejhorším případe se ale vykoná jeden průchod ve

funkci `_check_if_covered` a druhý ve funkci `_try_dont_care_values` za předpokladu, že všechny testy obsahují test se žolíkem. V reálném případě ale většina testů neobsahuje žolíka, no stále jde o značnou část testů, která se kontroluje 2-krát ve dvou bězích. Prvně pokládáme za fakt, že všechny kombinace pokryté v sade *ts* jsou již odstraněny ze sady kombinací π. Tímto je kontrola jestli je kombinace pokrytá zcela zbytečná. Pak nám postačuje funkce `_try_dont_care_values`, která jestli najde vhodný test na doplnění, nalezen test pozmění aby kombinaci pokryl. V opačném případě vytvoří test nový. Touto optimalizací by se měla doba běhů programu výrazně zkrátit. Z údajů z profilingu 3.1 to dělá cca. $\frac{1}{3}$ celkové doby běhu programu.

```
697     # iterate through param_combinations in pi set
698     for param_comb in pi_set:
699         # set temp_test_case values to default (to test_case)
700         temp_test_case = list(test_case)
701 
702         # transform param_comb into list
703         param_comb_list = [int(d) for d in str(param_comb)]
704 
705         # mask test_case
706         for j in range(0, len(test_case)):
707             if param_comb_list[j] == 0:
708                 temp_test_case[j] = 0
709 
710         # remove covered tuples
711         if tuple(temp_test_case) in pi_set[param_comb]:
712             pi_set[param_comb].remove(tuple(temp_test_case))
713 
714             # remove specific dont care values
715             if tc_index in dont_care_values:
716                 for i in range(0, len(temp_test_case)):
717                     if temp_test_case[i] == 0:
718                         continue
719                     if i in dont_care_values[tc_index]:
720                         dont_care_values[tc_index].remove(i)
721 
722                 if dont_care_values[tc_index] == []:
723                     del dont_care_values[tc_index]
724 
725         # note empty dictionary items (that does not contain any parameter value combinations)
726         if pi_set[param_comb] == set([]):
727             empty_indices_of_pi_set.append(param_comb)
```

Obrázek 3.2: Část funkce _remove_combinations odkazována v obr.3.1

.

Patrné množství času se tráví také při vytváření kopie seznamu, což je zajímavé z pohledu efektivity. Odstranění této kopie by mohlo potenciálně zvýšit efektivitu. Z kódu na obr. 3.2 jde vidět, že převod na seznam není potřeba. Úpravou řádků 703-708 na:

```
703         for j in range(0, len(test)case)):
704             if param_comb[j] == "0":
705                 temp_test_case[j] = 0
```

Obdobně i úpravou řádků 521-526 by se celkově zrychlil původní nástroj o přibližně $\frac{1}{16}$.

Ve funkci `_try_dont_care_values` jsou zbytečné kontroly jestli jsme na parametru který je v kombinaci.

```
828 for i in range(0, len(value_comb)):
829     if value_comb[i] == 0:
830         continue
831     # we cannot change the test if the value on position i is not
                                                        a don't care value
832     if (value_comb[i] != ts[index][i]) and (i not in
                                              dont_care_values[index]):
833         change_test_flag = False
834         break
```

Z tohoto kódu usuzujeme následující strukturu hodnoty ve `value_comb`: `[0,1,0,3]` kde „0“ reprezentuje parametr o který v dané kombinaci nemáme zájem a čísla „1“ a „3“ reprezentují pořadí hodnoty v bloku hodnot respektivního parametru. Pro „1“ to bude prvá hodnota v bloku v druhém parametru, pro „3“ zas třetí hodnota v bloku v čtvrtém parametru. U tohoto příkladu můžeme vidět neefektivitu hledání vhodného testu pro pokrytí kombinace. Jelikož jsou v příkladu dva zbytečné průchody smyčkou for, pro hodnoty „0“, tak by bylo vhodné se jich zcela zbavit. Aktuálně pro ověření zda test je nebo není vhodný pro pokrytí hledané kombinace se vykoná $t+n$ kontrol. Proto navrhujeme upravit strukturu `value_comb` jako mapu, kde klíčem bude kombinace parametrů, které kombinace obsahuje `[2,4]`. Takhle by byl reprezentován klíč pro daný příklad uveden výš v tomto paragrafu. Hodnotou v této mapě by bylo pole párů. První hodnota opomenutého páru by obsahovala indexy bloků v prvních $t-1$ parametrech zadaných v klíči: `[1]`. Druhá hodnota páru by bylo pole se všemi nepokrytými hodnotami pro poslední parametr v klíči: `[3]`. Výsledná struktura by pak vypadala takto:

```
[   //mapa
  [2,4]//klíč
    [//hodnota v mape - pole párů
      [1][3,5] // pár s dvěma nepokrytími hodnotami pro lepší představu
    ],
  [3,4]
    [
      [2][1,4,5],
      [3][2,3]
    ]
]
```

Při procházení takovouto strukturou by se zamezilo zbytečným kontrolám hodnot v poli kombinací které nejsou podstatné. Ve struktuře se vykoná nejvíc t průchodů smyčkou for pro jednu kombinaci. Tato struktura ale přichází s cenou zvýšené komplexity, zdrojů na ukládání do struktury a také zvýšená složitost přístupu ke konkrétní hodnotě v případe mazání ze sady kombinací π.

Po bližší prohlídce funkce *_remove_combinations* se zdá její implementace neefektivní, jelikož porovnává všechny nepokryté kombinace s každým testem z testovací sady, a následně při shodě nepokrytou kombinaci odstraňuje. Viz obr.3.2 řádky 698 a 711. Vhodnější by mohlo být si už při získávaní nejlepší hodnoty z bloků parametru pamatovat kombinace jenž se pokryjí. Pak stačí smazat právě tyto kombinace. Přichází to s cenou výpočetního a

paměťového nároku na vytváření nové proměnné, kde se budou držet všechny možnosti pro každý blok parametru.

Experimentální optimalizace

Jako jedinou optimalizaci z navrhovaných, jsme se rozhodli zkusit v původním řešení optimalizaci pro redukci vytváření kopií seznamů zmíněnou v návrhu 3.1. Důvodem pro výběr pravě této optimalizace byla její jednoduchost implementace.

	t=2	t=3	t=4
Původní verze	0.04s	1.6s	73.6s
Upravena verze	0.033s	1.2s	64s
Zrychlení	17.5%	25%	13%

Tabulka 3.1: Zobrazuje rozdíl v čase řešení sady testů pro vstup 10 parametrů o 10 blocích mezi původní a upravenou verzí

Z experimentální optimalizace můžeme vyvodit následující: vytváření kopie anebo přetypovaní má dopad na rychlost programu a proto se tomu při reimplementaci vyhneme, například zvolením vhodnější datové struktury. Dále se také vyhneme logickým chybám a nebudeme procházet pole vícekrát, když stačí jenom jednou.

Kapitola 4

Nová implementace

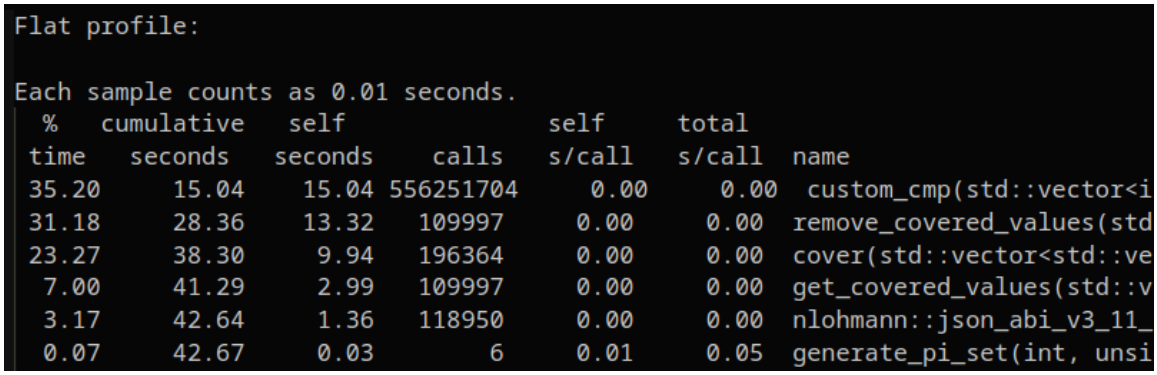
CombineNG je název nové implementace nástroje. Jde o konzolovou aplikaci, která na vstupu přijímá popis SUT ve formátu JSON souboru s obdobným formátem dat jako původní nástroj. Na výstupu vrací odpověď stejného JSON formátu, lehce odlišného od původního nástroje. Odpověď je složená z pořadí parametrů v testech, kvůli přeuspořádání parametrů pro efektivitu algoritmu, a samotné sady testů. Sada testů pokrývá všechny T-násobné kombinace bloků parametrů pro danou sílu testu, která může nabývat hodnot 1 až 6. „*Empirická data sesbírána institutem NIST a dalšími naznačuje, že softwarové chyby jsou způsobeny jenom pár proměnnými (6 nebo méně). Tohle zjištění má důležité implikace pro testování, protože naznačuje, že testování kombinací parametrů může poskytnout vysoce efektivní detekci chyb.*“[3] Pro testování pomocí testů nástroj umožňuje generovat v testech jak indexy tak konkrétní hodnoty. Umožňuje určit omezení, a tím zamezit pro daný systém nesmyslné, nebo nežádoucí kombinace bloků parametrů. Také umožňuje specifikovat hodnoty bloků parametrů pomocí výrazů jako předešlá implementace.

Implementovaný algoritmus zůstává IPOG-C, protože neměl negativní vliv na efektivitu předešlé implementace. Z toho důvodu budou v nové implementaci podobné funkce jako v původním nástroji plynoucí ze samotného algoritmu. Predešlá implementace zahrnovala i webové rozhraní, tím se ale tato bakalářská práce zabývat nebude. Reimplementace bude obsahovat implementaci algoritmu IPOG-C a specifikací hodnot parametrů výrazem.

První změna je výběr programovacího jazyka C++. Tento jazyk byl vybrán pro jeho schopnost využít více vláken, pro přístup k nízkým úrovním paměti v porovnání s jazykem Python, ale také kvůli pohodlným abstrakcím datových typů a práce s nimi v porovnání s jazykem C. První implementace algoritmu IPOG nám ukázala, jak velký je rozdíl v době běhu programu mezi těmito jazyky, viz tab. 4.1.

			Čas(s)		Testy	
t	parametry	hodnoty	Python	C++	Python	C++
2	10	10	0.0494804	0.001636	170	172
2	10	20	0.239012	0.010608	625	629
3	10	10	1.94437	0.17269	2390	2387
3	10	20	38.1142	5.13479	18214	18194
4	10	10	83.9707	30.6072	29769	29732
4	10	20	13121.3	8119.57	459668	461831

Tabulka 4.1: Výsledek porovnání testování původní a prvotní implementace nového nástroje na jednom vláknu



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 35.20     15.04    15.04 556251704     0.00     0.00  custom_cmp(std::vector<i
 31.18     28.36    13.32   109997     0.00     0.00  remove_covered_values(std
 23.27     38.30     9.94   196364     0.00     0.00  cover(std::vector<std::ve
  7.00     41.29     2.99   109997     0.00     0.00  get_covered_values(std::v
  3.17     42.64     1.36   118950     0.00     0.00  nlohmann::json_abi_v3_11_
  0.07     42.67     0.03        6     0.01     0.05  generate_pi_set(int, unsi
```

Obrázek 4.1: Výstup nástroje gprof při síle testu 4, počte parametrů 10 a každý parametr má 10 bloků

Druhou značnou změnou bude využití více vláken procesoru pro urychlení výpočtu. Místa kde je vhodné použít multithreading, jsme zjistili pomocí profilingu nástrojem gprof, viz obr. 4.1. Můžeme zde vidět tři funkce které zabírají nejvíc času, no jenom jedna je volaná mnohem víc krát než ty ostatní, `custom_cmp`. To naznačuje vhodné místo na paralelizmus kódu. Zbylé dvě zmíněné funkce nejsou vhodné pro paralelizování. Jeden z důvodů je poměr režie vůči získanému času v těchto funkcích, který je velmi nevhodný, protože se tyto funkce vykonají poměrně rychle. Druhým důvodem je, že se jejich funkcionalita jednoduše nedá dobře paralelizovat. Funkce `remove_covered_values` odstraňuje pokryté kombinace z π sady. Režie přístupu k této jedné proměnné, která se bude měnit za běhu, je mnohem vyšší, než zisk z paralelního přístupu. Jde o proměnnou, která efektivně dělá z této funkce jedno vláknovou operaci. Ve funkci `cover`, která hledá vhodného kandidáta z testů se žolíky pro pokrytí kombinace, by mohlo dojít za pomoci multithreadingu ke zrychlení. No samotná sada testů se žolíky je natolik malá, že režie více vláken převyšuje získaný čas. Žolík je neurčena hodnota v testech vytvořených při vertikálním rozšíření.

4.1 Vstup a výstup aplikace

Aplikace přijímá na vstupu zmiňovaný JSON formát, který vypadá následovně:

```
{
"name": "SUT_name",
"t_strength": "2",
"dont_care_values": "no",
"values": "values",
"parameters": [
    { "identificator": "a", "type": "integer", "blocks": ["'1'", "'3'"] },
    { "identificator": "b", "type": "enum", "blocks": ["42", "24", "84"] },
    { "identificator": "c", "type": "enum", "blocks": ["42", "24", "84"] },
    { "identificator": "d", "type": "string","blocks":["len<5","d='ts'"]}
],
"constraints": ["d.1 and !a.2"]
}
```

Obsahuje jméno systému pod testem, sílu testů t, „dont_care_values“ pro způsob jak doplnit testy se žolíky, „values“ neboli hodnoty ve kterých chceme mít testy reprezentovány,

parametry a nakonec omezení. Pole „dont_care_values“ určuje, zda se mají namísto žolíků doplnit náhodné nebo předem určené hodnoty, co znamená v tomto nástroji hodnotu z prvního bloku parametru je-li to možné vzhledem k omezením. V opačném případě se vybere hodnota z validního bloku. Parametry jsou označeny identifikátorem, mají svůj typ dle kterého mohou být bloky různě specifikovány (viz 4.2).

```
{
"name":"SUT_name",
"identificators":["b","c","a","d"],
"tests":[
  ["42","42","1","5377"],
  ["42","24","1","ts"],
  ["42","84","1","5377"],
  ["24","42","1","ts"],
  ["24","24","1","5377"],
  ["24","84","1","ts"],
  ["84","42","1","5377"],
  ["84","24","1","ts"],
  ["84","84","1","5377"],
  ["42","42","3","5377"],
  ["42","42","3","ts"]
  ]
}
```

Na ukážce výstupu, pro vstup výše, vidíme jméno systému pod testem, pole „identificators“ znázorňující pořadí parametrů v testech, jelikož ipog-c algoritmus vyžaduje jejich přeuspořádáni od parametru s nejvyšší četnosti bloku po nejnižší a samotné testy ve vyžádané podobě.

4.2 Specifikace bloků

Nakolik jde o reimplementaci předešlé implementace, tak specifikace jazyku, gramatik a pravidel přepisu byli převzaty z původní aplikace z těchto zdrojů:

- webový nástroj na https://combine.testos.org/
- zdrojové kódy na https://pajda.fit.vutbr.cz/testos/combine-bcc
- Bakalářská práce ČERVINKA, Radim. Asistent pro generování testovacích scénářů. [2]

Specifikace typu integer

Definice tohoto typu byla převzata úplně z [2]. Při implementaci jsme se drželi gramatiky původního nástroje. Gramatika obsahuje terminály AND, OR, LPAREN, RPAREN, LE, LT, GE, GT, EQ, NEQ, INT, VAR, STRING. Pro gramatiku viz tabulka 4.2 a pro pravidla přepisu viz tabulka 4.3. Příklady specifikace:

- ”42”
- $i = 42$

STRING	představuje základní popis bloku
AND	logická spojka and (psána *and*)
OR	logická spojka or (psána *or*)
LPAREN	levá kulatá závorka (psána *(*)
RPAREN	pravá kulatá závorka (psána *)*)
VAR	identifikátor parametru, ve kterém se blok nachází blok
INT	celé číslo
LT	komparační operátor menší než (psáno <)
LE	komparační operátor menší nebo rovno (psáno <=)
GT	komparační operátor větší než (psáno >)
GE	komparační operátor větší nebo rovno (psáno >=)
EQ	komparační operátor rovná se (psáno =)
NEQ	komparační operátor nerovná se (psáno *!*=)

Tabulka 4.2: Popis terminálů v gramatice jazyka pro popis bloku typu integer[2].

S → STRING
S → E
E → E AND E
E → E OR E
E → VAR op INT
E → LPAREN E RPAREN

Tabulka 4.3: Seznam přepisovacích pravidel v gramatice jazyka pro popis bloku typu integer (značka op zde zastupuje skupinu terminálů LT, LE, GT, GE, EQ a NEQ)[2].

- $i < 30\ and\ (i > 15\ and\ i! = 20)$
- $i <= 30\ or\ i = 50$

Specifikace typu float

Definice tohoto typu byla převzata úplně z [2]. Při implementaci jsme se drželi gramatiky původního nástroje. Gramatika obsahuje terminály AND, OR, LPAREN, RPAREN, LE, LT, GE, GT, EQ, NEQ, NEGINF, INF , NAN_ ,FLOAT, VAR, STRING. Pro gramatiku viz tabulka 4.4 a pro pravidla přepisu viz tabulka 4.5. Příklady specifikace:

- ”42.0”
- $f = 42.24$
- $f < 3.9\ and\ (f > 1.5\ and\ f! = 2)$
- $f <= 0.3\ or\ f = 5$

Specifikace typu date

Definice tohoto typu byla převzata téměř úplně z [2]. Gramatika pro specifikaci tohoto typu obsahuje stejné terminály i přepisovací tabulky jako specifikace typu integer s jediným

STRING	představuje základní popis bloku
AND	logická spojka and (psána *and*)
OR	logická spojka or (psána *or*)
LPAREN	levá kulatá závorka (psána *(*)
RPAREN	pravá kulatá závorka (psána *)*)
VAR	identifikátor parametru, ve kterém se blok nachází blok
FLOAT	desetinné číslo
INF	nekonečno (psáno *inf*)
NEGINF	mínus nekonečno (psáno *-inf*)
NAN_	Not A Number (psáno *nan*)
LT	komparační operátor menší než (psáno <)
LE	komparační operátor menší nebo rovno (psáno <=)
GT	komparační operátor větší než (psáno >)
GE	komparační operátor větší nebo rovno (psáno >=)
EQ	komparační operátor rovná se (psáno =)
NEQ	komparační operátor nerovná se (psáno *!*=)

Tabulka 4.4: Popis terminálů v gramatice jazyka pro popis bloku typu float[2].

S → STRING
S → VAR EQ INF
S → VAR EQ NEGINF
S → VAR EQ NAN
S → E
E → E AND E
E → E OR E
E → VAR op FLOAT
E → LPAREN E RPAREN

Tabulka 4.5: Seznam přepisovacích pravidel v gramatice jazyka pro popis bloku typu float (značka op zde zastupuje skupinu terminálů LT, LE, GT, GE, EQ a NEQ)[2].

rozdílem, terminál INT je nahrazen terminálem DATESTRING ve formátu YYYY-MM-DD. Formát data byl pozměněn tak aby zodpovídal realitě nástroje na combine.testos.org/. Příklady specifikace:

- "$1970 - 1 - 1$"
- $d = 2024 - 3 - 1$
- $d! = 1970 - 11 - 11$ *and* $d > 2000 - 01 - 01$
- $d > 1970 - 1 - 1$ *and* $(d < 2000 - 1 - 1$ *or* $d > 2100 - 12 - 2)$

Specifikace typu time

Definice tohoto typu byla převzata úplně z [2]. Gramatika je podobně jako i u typu date sdílená s typem integer. Znovu je jediným rozdílem terminál INT, který je nahrazen terminálem TIMESTRING ve formátu HH:MM:SS. Příklady specifikace:

- "$23 : 00 : 00$"
- $t = 23 : 0 : 0$
- $t! = 09 : 24 : 13$ *and* $t > 9 : 25 : 15$
- $t > 00 : 2 : 13$ *and* $(t < 9 : 24 : 09$ *or* $t > 21 : 10 : 2)$

Specifikace typu string

Definice byla převzata z [2] a mírně upravena. Byl odstraněn L neterminál, nakolik nijak neovlivňoval logiku výstupu ale za to omezoval uživatele. Při řešení SAT/SMT problémů se výraz beztak zjednoduší a použije se poskytnutá hodnota, pro kterou má problém řešení. Při implementaci jsme se drželi gramatiky původního nástroje. Gramatika obsahuje terminály AND, OR, LPAREN, RPAREN, LE, LT, GE, GT, EQ, NEQ, LEN, CAT, ALPHANUM, ALPHA, NUM, ONEOF, INT, VAR, STRING. Pro gramatiku viz tabulka 4.6 a pro pravidla přepisu viz tabulka 4.7. Terminály které nejsou v tabulce 4.6 jsou stejné jako v předešlých gramatikách. Změny v této implementaci jsou ve volnějších pravidlech přepisu povolující použít závorky na jakýkoliv platný výraz. Příklady specifikace:

- "$oneof(aasdahoj, janko, hruska)$"
- $s = '!@\$!\ \%\ !,'$
- $(len < 6$ *and* $category = alphabetic)or(len >= 15$ *and* $len <= 20$ *and* $category = num)$
- $category = num$ *and* $len < 3$

LEN	slouží pro specifikaci délky řetězců v bloku (psáno *len*), používá se s komparačními operátory
CAT	určení kategorie stringů (psáno *category*)
ALPHNUM	kategorizace řetězců v bloku, určuje, že budou obsahovat písmena a číslice (psáno *alphanum*)
ALPHA	kategorizace řetězců v bloku, určuje, že budou obsahovat pouze písmena (psáno *alphabetic*)
NUM	kategorizace řetězců v bloku, určuje, že budou obsahovat pouze číslice (psáno *num*)
ONEOF	možnost výčtu možných hodnot v bloku (psáno *oneof()*, v závorce seznam hodnot oddělených čárkou)

Tabulka 4.6: Popis vybraných terminálů v gramatice jazyka pro popis bloku typu string[2].

S → STRING
S → ONEOF
S → E
E → E AND E
E → E OR E
E → VAR EQ STRING
E → VAR NEQ STRING
E → LEN op INT
E → LPAREN E RPAREN
E → CAT EQ ALPHNUM
E → CAT EQ ALPHA
E → CAT EQ NUM

Tabulka 4.7: Seznam přepisovacích pravidel v gramatice jazyka pro popis bloku typu string (značka op zde zastupuje skupinu terminálů LT, LE, GT, GE, EQ a NEQ)[2].

LEN	slouží pro specifikaci délky emailu (psáno *len*), používá se s komparačními operátory
DOMAIN	přiřazení hodnoty, které se rovná nebo nesmí rovnat doména emailu (psáno *domain*)
DOMAINLEN	specifikuje délku doménové části emailu (psáno *domain_len*)
LOCALPART	přiřazení hodnoty, které se rovná nebo nesmí rovnat lokální části emailu (psáno *local_part*)
LOCALPARTLEN	specifikuje lokální část emailu (psáno *local_part_len*)
DOMAIN_STRING	představuje popis doménové části bloku typu email
LOCAL_PART_STRING	představuje popis lokální části bloku typu email

Tabulka 4.8: Popis vybraných terminálů v gramatice jazyka pro popis bloku typu email[2].

S → STRING
S → E
E → E AND E
E → LEN op INT
E → LOCALPART EQ LOCAL_PART_STRING
E → LOCALPART NEQ LOCAL_PART_STRING
E → DOMAIN EQ DOMAIN_STRING
E → DOMAIN NEQ DOMAIN_STRING
E → DOMAINLEN op INT
E → LOCALPARTLEN op INT

Tabulka 4.9: Seznam přepisovacích pravidel v gramatice jazyka pro popis bloku typu email (značka op zde zastupuje skupinu terminálů LT, LE, GT, GE, EQ a NEQ)[2].

Specifikace typu email

Při implementaci jsme se drželi gramatiky původního nástroje. Gramatika obsahuje terminály AND, LPAREN, RPAREN, LE, LT, GE, GT, EQ, NEQ, LEN, DOMAIN, DOMAIN_LEN,LOCAL_PART, LOCAL_PART_LEN, VAR, INT a přidané terminály DOMAIN_STRING, ve formátu A.A a LOCAL_PART_STRING ve formátu A, kde A je libovolný alfanumerický znak. Přidány terminály slouží pro detailnější kontrolu vstupu pro jednotlivé části emailové adresy. Pro gramatiku viz tabulka 4.8 a pro pravidla přepisu viz tabulka 4.9. Definice byla převzata z [2] a mírně upravena. Příklady specifikace:

- $len > 3$ *and* $local\_part\_len = 4$ *and* $domain\_len = 4$
- $domain =' as.c'$ *and*$(local\_part =' 1234')$
- $'marienka.alienka@testos.com'$
- $domain\_len = 5 and local\_part =' jozko'$

Specifikace typu phonenum

Definice tohoto typu byla převzata z [2] a mírně upravena. Při implementaci jsme se drželi gramatiky původního nástroje. Gramatika obsahuje terminály AND, LPAREN, RPAREN,

NUMBER	slouží pro specifikaci té části telefonního, která se nachází za prefixem (psáno *number*)
NUMBERLEN	slouží pro specifikaci délky té části telefonního, která se nachází za prefixem (psáno *len*)
PREFIX	specifikuje mezinárodní prefix telefonního čísla (psáno *prefix*)
NUM_STRING	představuje popis číselné části bloku typu phonenum
PREFIX_STRING	představuje popis prefixové části bloku typu phonenum

Tabulka 4.10: Popis vybraných terminálů v gramatice jazyka pro popis bloku typu phonenum[2].

S → STRING
S → E
E → E AND E
E → PREFIX EQ PREFIX_STRING
E → PREFIX NEQ PREFIX_STRING
E → NUMBER EQ NUM_STRING
E → NUMBER NEQ NUM_STRING
E → NUMBERLEN op INT

Tabulka 4.11: Seznam přepisovacích pravidel v gramatice jazyka pro popis bloku typu phonenum (značka op zde zastupuje skupinu terminálů LT, LE, GT, GE, EQ a NEQ)[2].

LE, LT, GE, GT, EQ, NEQ, LEN, NUM, PREFIX, INT, STRING, VAR a přidané terminály PREFIX_STRING, ve formátu +D{3} a NUM_STRING ve formátu D{7-12}, kde D je libovolný numerický znak a číslo v závorkách reprezentuje počet těchto znaků. Přidány terminály slouží pro detailnější kontrolu vstupu pro jednotlivé části telefonního čísla. Pro gramatiku viz tabulka 4.10 a pro pravidla přepisu viz tabulka 4.11. Příklady specifikace:

- $number = '12345678901'\ and\ (prefix =' +420')$
- $number\_len = 9$
- $len > 3\ and\ local\_part\_len = 4\ and\ domain\_len = 4$

Specifikace typu id

Definice tohoto typu byla úplně převzata z [2]. Při implementaci jsme se drželi gramatiky původního nástroje. Gramatika obsahuje terminály AND, EQ, DELIM, BLKLEN, BLKCNT, CHAR, INT, STRING, VAR. Gramatika pro nové terminály DELIM, BLKLEN, BLKCNT a CHAR je v tabulce 4.12. Pro pravidla přepisu viz tabulka 4.13. Příklady specifikace:

- $block\_len = 6\ and\ block\_count = 3\ anddelimiter =' -'$
- $block\_count = 2$
- $id =' 1234.1234.1234'$

CHAR	jeden znak, slouží pro specifikaci oddělovače
DELIM	delimiter nebo také oddělovač bloků v ID (psáno *delimiter*)
BLKLEN	specifikuje délku bloku (psáno *block_len*)
BLKCNT	specifikuje počet bloků v ID (psáno *block_count*)

Tabulka 4.12: Popis vybraných terminálů v gramatice jazyka pro popis bloku typu id[2].

S → STRING
S → E
E → E AND E
E → VAR EQ STRING
E → VAR NEQ STRING
E → DELIM EQ CHAR
E → BLKCNT EQ INT
E → BLKLEN EQ INT

Tabulka 4.13: Seznam přepisovacích pravidel v gramatice jazyka pro popis bloku typu id (značka op zde zastupuje skupinu terminálů LT, LE, GT, GE, EQ a NEQ)[2].

Specifikace omezení

Definice tohoto typu byla úplne převzata z [2]. Při implementaci jsme se drželi gramatiky původního nástroje. Gramatika obsahuje terminály AND, EQ, DELIM, BLKLEN, BLKCNT, CHAR, INT, STRING, VAR. Gramatika pro nové terminály DELIM, BLKLEN, BLKCNT a CHAR je v tabulce 4.14. Pro pravidla přepisu viz tabulka 4.15.

4.3 Použité typy proměnných

V algoritmu IPOG jde především o procházení prvkami v množinách od prvního mnohokrát po poslední. Nachází se zde i přístup do množin na i-té pozici v případe mazání pokrytých kombinací z množiny nepokrytých kombinací. Jelikož se ve velké části algoritmu pracuje s množinami sekvenčně, zvolili jsme proto jako hlavní proměnnou typ `std::vector<>` pro její rychlí sekvenční přístup.

Základním typem se tak stal `std::vector<int>` (pro jednoduchost `matrix_row`), který reprezentuje samotný test, indexy do bloku v parametru, indexy parametrů, nepokryté

BLOCK	specifikace konkrétního bloku konkrétního parametru (psáno *identifikátor_parametru.číslo_bloku*)
AND	logická spojka and (psáno *and*)
OR	logická spojka or (psáno *or*)
XOR	logická spojka xor (psáno *xor*)
IMPL	implikace (psáno ->)
EQUIV	ekvivalence (psáno <->)
NOT	negace(psáno *!*)
LPAREN	levá kulatá závorka (psáno *(*)
RPAREN	pravá kulatá závorka (psáno *)*)

Tabulka 4.14: Popis vybraných terminálů v gramatice jazyka pro popis omezení[2].

S → E
E → E AND E
E → E OR E
E → E XOR E
E → E IMPL E
E → E EQUIV E
E → NOT E
E → LPAREN E RPAREN
E → BLOCK

Tabulka 4.15: Seznam přepisovacích pravidel v gramatice jazyka pro popis omezení[2].

bloky v parametru, atd. Výsledné pole testů *ts* má typ `std::vector<matrix_row>` (pro jednoduchost `matrix`).

Dále je zde proměnná *ps* reprezentující parametry a jejich bloky. Popis parametru se skládá z jeho jména a vygenerovaných hodnot pro každý blok dle specifikací typů v části 4.2. Typ parametru je tehdy `std::pair<std::string,matrix_row>` (pro jednoduchost `params_row`). Pole těchto parametrů má pak typ `std::vector<params_row>` (pro jednoduchost `params_matrix`). V podstatě jde o mapu s párem <klíč, hodnota>, no je vhodnější pro sekvenční přístup, který se zde používá.

Do proměnné typu `std::pair<matrix_row,z3::expr>` (pro jednoduchost `constr_t`) jsou ukládáni výrazy reprezentující omezení. Ve `constr_t` se agregují unikátní zahrnuté parametry omezení v prvním prvku páru a samotná omezení v druhém prvku. Omezení se pak na základe společného prvku sloučí pomocí logické spojky `&&` neboli `AND` do jednoho omezení v druhém prvku páru. Unikátní zahrnuté parametry se rozšíří o unikátní parametry druhého omezení tak, aby v zahrnutých parametrech byl stále každý parametr unikátní. Vzniká tak proměnná skupin omezení *constraints* typu `std::vector<constr_t>`.

Proměnná nepokrytých kombinací *pi_set* (v 2.3 jako π) má celkem komplexní strukturu ze 2 důvodů. Prvním důvodem je snížení počtu porovnání při kontrole, zda test je vhodný kandidát pro pokrytí dané kombinace, viz 3.1, třetí paragraf. Druhým důvodem je vhodnější rozdělení do jednotlivých úkolů pro vícevláknové zpracování. Strukturu lze popsat zevnitř následovně: prvek nejvíc zanořený se skládá ze dvou částí typu `matrix_row`, kde první obsahuje indexy bloků parametrů 0 až n-1 a druhá obsahuje indexy bloků parametru n, které spolu tvoří pár typu `std::pair<matrix_row, matrix_row>` (pro jednoduchost jako `pi_elem`). Informace kterým parametrům patří indexy bloků je přidána typem `std::pair<matrix_row, std::vector<pi_elem>`. Zde `matrix_row` reprezentuje indexy parametrů do kterých se odkazují indexy bloků z `pi_elem`. Jelikož pro jednu kombinaci indexů parametrů může být nepokrytých více kombinací indexů bloků, tak jsou `pi_elem` vložený do `std::vector`. Takhle vypadá jeden prvek množiny *pi_set*, která má tedy typ `std::vector<std::pair<matrix_row, std::vector<pi_elem>>>` (pro jednoduchost `pi_set_type`).

Odstraňování pokrytých kombinací bylo v předešlé implementaci analyzováno jako místo, kde se tráví značná část doby běhu programu. Proto jsme vytvořili proměnnou *to_rm*, která si uchovává všechny pokryté kombinace pro každý blok v aktuálně přidávaném parametru v horizontálním rozšíření. Tato proměnná se před implementací omezení používala také pro zjištění nejvhodnějšího bloku pro rozšíření, jednoduše získáním indexu prvku s nejvíc kombinacemi na smazání. Avšak omezení zavedli novou možnost: pro kombinaci bloku s daným

testem nemá omezení řešení. Tento fakt není možno reprezentovat v této proměnné, a proto se pro tento účel používá pomocná proměnná. Opomenuta proměnná *to_rm* má tedy typ `std::vector<std::vector<std::pair<int, int>>>`, kde první integer v páru je index do první úrovně množiny *pi_set* a druhý integer je index do druhé úrovně té samé množiny. Jako index bloku do parametru se použije pořadí daného vektoru párů v proměnné *to_rm*.

V části kódu provádějící vertikální růst se využívá množiny žolíků *joker_set* kvůli rychlému přístupu ke konkrétnímu testu z proměnné *ts*. Zajistí se tím minimální počet kontrol nutný ke zjištení zda existuje test, kterého doplněním by se pokryla daná kombinace. Tato proměnná je typu `std::vector<std::pair<int,int>>`, kde pár reprezentuje dvojice index do pole testů a počet žolíků v daném testu. V případě odstranění posledního žolíka z testu se odstraňuje i pár v množině žolíků.

4.4 Multithreading

Multithreading je způsob jak rozdělit logickou strukturu provádění kódu, rozdělit práci a získat tak výsledek rychleji nebo si prodloužit dobu běhu programu v případe nevhodného použití. Programování na více vláknech je obtížné hned z několika důvodů:

- Výběr vhodné části kódu k paralelizaci
- Synchronizace dat
- Obtížný debuging

Přesto že multithreading má nevýhody, ho používáme v nové implementaci pro hlavní výhodu, kterou je zkrácení doby výpočtu výsledku. Algoritmus IPOG-C je sekvenční, no existuje v něm místo pro vhodnou paralelizaci, i když velice omezené. Na toto místo poukazuje i profiling 4.1 na jednom vláknu, který jako funkci, v které se strávilo nejvíc času, vyhodnotil `custom_cmp` nacházející se ve funkci `get_covered_values`. Úkolem funkce `get_covered_values` je získat blok z následujícího parametru, který pokryje co nejvíc nepokrytých kombinací v „pi_set“ proměnné. Za tímto cílem se prochází všechny nepokryté kombinace a porovnávají se s daným testem, jako argument do funkce, jestli může test tuto kombinaci pokrýt. Porovnaní mezi kombinací a testem se dělá v opomenuté funkci `custom_cmp`. Toto porovnání bylo zjednodušeno o $n - t$ porovnání, kde n je celková délka testu a t je síla testu, díky implementaci návrhu na zlepšení 3.1 struktury mapy. Navzdory zjednodušení se ve funkci `custom_cmp` stále trávilo mnoho času. Tahle funkce se tehdy implementovala, tak aby využívala více vláken. Po prvotní změně se doba běhu výrazně zhoršila. Důvodem byla velká režie obrovského počtu volání do této funkce, řádově 100 miliony volání, viz 4.1 sloupec „calls“. Pak jsme aplikovali celkem běžnou praktiku zvanou „batching“, neboli vytváření balíků s informacemi pro vyhodnocení více volání funkce v jednom vláknu. Režie se značně snížila, no stále byla vysoká. Aplikujeme další také celkem běžnou praktiku, vytvořili jsme „thread pool“. Je to třída s abstrakcí nad samotnými vlákny (anglicky threads), která si vytvoří počet vláken dle počtů jader v procesoru a přijímá úkoly, které vlákna postupně zpracují. Tímto se snížila režie vytváření vláken pro každý batch. Paralelizmus ale pokračuje dále než jen jedno porovnání. Po úspěchu porovnání se také zkontroluje platnost kombinace a testu, upraví se test, aktualizuje se počítadlo počtu pokrytých kombinací pro respektivní blok parametru. Tady nastává další problém spojen s multithreadingem, synchronizace dat. Tento problém nastává až v okamžiku kdy se data, ke kterým přistupuje více vláken, přepíšou jedním z vláken bez exkluzivního přístupu. Proto se po úspěšném porovnání zamkne zámek (anglicky *lock* nebo *mutex* odvozeno

od *mutual exclusion*) pro daný blok, provedou se víš opomínané akce a následně se zámek opět odemkne. Takhle jsou data synchronizovaná. Multithreading kvůli své podstatě nezaručuje determinizmus, protože úkol zadán později může být vypočten dříve než, úkol zadán před ním. Nová implementace algoritmu je tehdy nedeterministická, no testování ukázalo, že odchylka je poměrně malá. Nová implementace se stala nedeterministickou kvůli testům se žolíky. Jako příklad mějme test `[-1,2,5]` a test `[-1,2,3]`, a chceme pokrýt kombinaci `[0,3][5,2]`. Výsledný test z kombinace je `[5,-1,-1,2]` a vyhovují jí oba testy, no výsledek závisí na tom, které vlákno dostane zámek nad blokem 2. Tímto způsobem se vnesla jenom velmi malá odchylka v počtu výsledných testů do nové implementace.

4.5 Pomocné knižnice a jiný software

Nová implementace používá 2 nestandardní C/C++ knižnice. Jednu pro čtení JSON formátu na vstupu a jeho zápis na výstupu. Druhou pro řešení splnitelnosti booleovské formule (anglicky Boolean Satisfiability Problem, zkratka SAT) a splnitelnosti modulo teorie (anglicky Satisfiability Modulo Theories, zkratka SMT). Pro parsování omezení a specifikací hodnot parametrů by bylo vhodnější použít nástroje parsující na základe gramatik a precedencí, než si to implementovat. Jenže v C++ jsme nenašli takový nástroj s jednoduchým použitím. Kvůli relativní jednoduchosti gramatik a pravidel pro omezení, složitosti existujících nástrojů a jednoduchou specifikaci jednotlivých typů hodnot jsme si parser implementovali sami.

Práce s JSON formátem

Pro zpracování JSON vstupu byl zvolen nlohmann/json[1], který v porovnání s jinými nástroji není nejrychlejší, no má jednoduché používání a také stačí do projektu zahrnout jediný hlavičkový soubor pro jeho použití. Další alternativy byli:

- Vlastní implementace práce s tímto formátem
- RAPIDjson
- simdjson

Výběr právě tohoto softwaru pro práci s JSON souborem může ovlivnit efektivitu kódu, avšak nepatrně nakolik se používá jenom na čtení vstupu a zápis na výstup pouze jednou.

SAT/SMT solver

Problém ověření validity testů a získaní konkrétní hodnoty pro kterou platí omezení, nebo specifikace hodnoty není vůbec triviální. Proto jsme si pomohli celkem známým a rozšířeným SMT solvrem Z3 od společnosti Microsoft. SMT solver je schopný řešit SAT problémy jako jsou omezení, jelikož SAT je podmnožinou SMT. Pro řešení jenom omezení by nám stačil SAT solvr, no pro zachování funkcionality původního nástroje Combine je zapotřebí i SMT solver pro získání konkrétní hodnoty splňující specifikaci hodnoty parametru.

Parser

V reimplementaci je napsán vlastní lexer a parser. Důvodem byli v celku jednoduché gramatiky, přepisové tabulky a aplikace třetích stran, které byli buď složité na použití, nebo

s nevhodným rozhraním. Kvůli tomu bylo jednodušší napsat si vlastní. Funkce zastupující lexer přijímá jako argumenty řetězec a regulární výrazy(vytvořených na základe gramatiky), dle kterých se má analyzovat řetězec. Výstupem je pole dvojic, kde první je ve dvojici řetězec, který zodpovídá tokenu v druhé části.Funkce parseru přijímá jako argumenty výstup z víš opomínaného lexeru a vytvoří výraz dle přepisovacích pravidel.

Kapitola 5

Porovnání původní a nové implementace

Porovnání bylo provedeno ve třech dimenzích. První byla čas výpočtu sady testů, čím kratší doba výpočtu, tím lepší je implementace. Druhá dimenze byla velikost výsledné sady testů, čím menší, tím lepší. Poslední třetí dimenze byla paměťoví nárok implementace, čím menší, tím lepší.

Testování pro prvou a druhou dimenzi bylo prováděno na vstupních sadách pro sílu t od 2 po 4, pro 10 až 20 parametrů, kde každý parametr má 10 až 20 bloků hodnot. Výjimkou je test síly t 4, 20 parametrů o 20-ti blocích každý, kvůli extrémní časové náročnosti pro obě verze nástroje. Z tabulky 5.1 vidíme, že nová implementace nástroje přinesla určité zrychlení při zachování přibližně rovnaného počtu výsledných testů. Největší zrychlení 20 až 45 krát vidíme na příkladech se vstupní sílou 2, tedy vstupy generující do 1000 testů. Rychlost v této části přisuzujeme samotnému jazyku C++ a jeho minimální režii, díky které, jak se ukázalo je o hodně rychlejší. Střední zrychlení 6 až 12 krát v příkladech se vstupní sílou 3, vstupy generující řádově 10 000 testů. Zmenšení zrychlení v druhé části považujeme za přirozené vzhledem ke zvýšené náročnosti testů a také režii samotného jazyka s tímto spojené. Předpokládáme, že režie v jazyku python byla příliš velká v první části, a proto neutrpěl tolik na rychlosti. Nejmenší až téměř žádné zrychlení pozorujeme u příkladů se vstupní sílou 4, vstupy generující řádově 100 000 testů. Zde to přisuzujeme ne zcela vhodné implementaci multithreadingu v nové implementaci, a také dosažení limitů paměti rezervované pro proměnné a jejich realokaci. Ve výsledku je ale nová implementace stále rychlejší než ta původní i když jenom o chlup v některých případech.

Třetí dimenzi jsme testovali na podobné sade vstupů, viz tabulka 5.2. Z této tabulky vidíme zvýšení v prvotní paměťové náročnosti u nové implementace. Rozdíl mezi paměťovými nároky naznačuje vyšší efektivitu z pohledu paměti u nové implementace, co pozorujeme zejména v posledních dvou řádcích pro sílu 4, 10 parametrů, 10 hodnot a 15 parametrů, 15 hodnot. V původní implementaci je nárůst značně vetší než v nové implementaci nástroje. Tomuto faktu naznačuje i sloupec „zvýšení nároku paměti X-krát“, kterého hodnoty v tabulce směrem dolů klesají. V případe, že by se paměťoví nárok u obou verzí nástroje měnil rovnaným způsobem, by byl tenhle sloupec téměř konstantní hodnotu, ale není.

Jelikož oba nástroje implementují omezení, provedli jsme testy všech dimenzí i s takovouto variantou vstupů ,viz tabulka 5.3. Při testování omezení jsme narazili na chybu implementace v jazyku python, která způsobovala existenci nesprávných testů ve výsledné sade. Tuto chybu možno pozorovat u kombinace více omezení a speciálně omezení $e.3$ *and* $!j.1$ se

			Čas(s)			Testy		
t	parametry	hodnoty	Python	C++	zrychlení X-krát	Python	C++	Δ%
2	10	10	0.05	0.02	3.02	170	172	1.18%
		15	0.12	0.01	22.27	364	358	-1.65%
		20	0.24	0.01	22.87	625	622	-0.48%
	15	10	0.15	0.00	35.24	193	193	0.00%
		15	0.38	0.01	31.68	417	416	-0.24%
		20	0.72	0.02	29.65	712	713	0.14%
	20	10	0.32	0.01	45.47	217	216	-0.46%
		15	0.82	0.02	40.36	457	457	0.00%
		20	1.64	0.04	37.26	786	790	0.51%
3	10	10	1.94	0.22	8.84	2390	2384	-0.25%
		15	10.47	1.81	5.77	7810	7786	-0.31%
		20	38.11	5.78	6.59	18214	18046	-0.92%
	15	10	9.35	1.15	8.16	3008	3009	0.03%
		15	43.21	5.98	7.23	9926	9872	-0.54%
		20	144.23	20.97	6.88	23108	22989	-0.51%
	20	10	30.68	2.50	12.29	3459	3453	-0.17%
		15	130.31	13.68	9.53	11476	11449	-0.24%
		20	404.97	50.87	7.96	26807	26702	-0.39%
4	10	10	83.97	22.46	3.74	29769	29511	-0.87%
		15	1536.73	455.68	3.37	147906	146413	-1.01%
		20	13121.30	3746.53	3.5	459668	457673	-0.43%
	15	10	499.47	170.70	2.93	40549	40377	-0.42%
		15	5566.69	3629.33	1.53	202411	201912	-0.25%
		20	43152.20	31718.10	1.36	631732	631809	0.01%
	20	10	2146.45	710.88	3.02	48636	48415	-0.45%
		15	17148.70	14397.40	1.19	243330	243015	-0.13%
		20	-	-	-	-	-	-

Tabulka 5.1: Výsledek testování původního a nového nástroje, SUT bez omezení

			Python		C++		
t	parametry	hodnoty	Paměť	Δn	Paměť	Δn	zvýšení nároku paměti X-krát
2	5	5	5228	-	63116	-	11.07
		10	5504	5.28%	65324	3.50%	10.87
	10	5	5292	-3.85%	63336	-3.04%	10.97
		10	5316	0.45%	67256	6.19%	11.65
3	5	5	5148	-3.16%	63356	-5.80%	11.31
		10	5584	8.47%	67920	7.20%	11.16
	10	5	5596	0.21%	63472	-6.55%	10.34
		10	6212	11.01%	65556	3.28%	9.55
4	5	5	5336	-14.10%	63680	-2.86%	10.93
		10	7652	43.40%	67044	5.28%	7.76
	10	5	5968	-22.01%	65252	-2.67%	9.93
		10	15876	166.02%	85160	30.51%	4.36
	15	15	89160	461.60%	370616	335.20%	3.16

Tabulka 5.2: Výsledek testování paměťových nároků původního a nového nástroje, SUT bez omezení, sloupec „Paměť“ reprezentuje velikost maximálního rezidenčního setu v kilobytech. Δn je rozdíl mezi paměťovým nárokem z předchozího řádku tabulky a aktuálním.

zdá, že má značný vliv na způsobení této chyby. Z hlediska paměťových nároků byli obě verze na své úrovni, kterou si drželi pro všechny počty omezení s minimální odchylkou. Pro Combine to bylo přibližně 5MB a pro CombineNG přibližně 36MB. Z tohoto důvodu jsme je v tabulce neuváděli. Nová implementace v porovnání mezi testem s dvěma omezeními a třemi výrazně zpomalila. Její doba běhu je téměř 8-krát větší zatím co u původní implementace je tento rozdíl méně než 2-krát. Zde se může ukazovat nevhodná implementace aplikace omezení u nového nástroje, anebo také to může být důsledkem nesprávné implementace v původní implementaci, které neprovádí všechny kontroly mezi omezeními a testy. Jelikož nemůžeme s jistotou říct, zda je chybný původní nebo nový nástroj, aplikování omezení je proto vhodné místo ke zlepšení a důslednějšímu testování.

Nedostatky a budoucí vylepšení

V této části se budeme věnovat nedostatkům a chybám nové implementace a návrhů jak je odstranit. Za první nedostatek považujeme náš vlastní lexer a parser, z důvodu slabého testování této části nakolik to nebylo hlavním cílem této bakalářské práce a z důvodu příliš dlouhého jediného souboru „solver_wrapper.cpp“ cca 2000 řádků. Navrhujeme proto refaktorizaci jak daného souboru, tak samotného lexeru a parsru.

Dále se při testování objevila chyba v generování řetězce bloku typu string v případě „s != ’text’“. Výsledný řetězec sice splňuje tuto podmínku, no je prázdný. Tím je nesplněna definice, která zmiňuje řetězec o délce 2 nebo větší. Další návrh na budoucí práci je tedy bližší náhled na tuto chybu a její opravu.

Ve výsledcích testování nás nemile překvapil úpadek v zrychlení v porovnání s původní implementací. To může naznačovat nevhodné manipulování s daty, nevhodné struktury, ukládání dat nevhodně nebo ukládání nesprávných dat. Také to může naznačovat neefektivní implementaci multithreadingu, nebo práci s daty v ní anebo úplně jiné příčiny,

	Testy			Čas(s)		
Počet omezení	Python	C++	Δ%	Python	C++	
4	111	31	-72.07%	3.86	11.81	0.33
3	112	112	0.00%	2.29	5.18	0.44
2	94	94	0.00%	1.35	0.69	1.96
1	94	94	0.00%	1.17	0.52	2.25

Tabulka 5.3: Výsledek porovnání testování s omezeními v původním a novém nástroji, Δ% je rozdíl mezi původní a novou implementací v počte testů a časech respektive. Byl použit vstup viz 5.1, s posloupným ubíráním omezení od posledního.

kterých jsme si není vědomy. Proto navrhujeme analýzu a zvýšení efektivity nástroje i v této části.

Jako poslední nedostatek jsme pozorovali delší počítání sady testů pro vstup s omezeními. Příčinou by mohlo být nesprávné použití knihovny Z3, zvolení nevhodné knihovny jako také nebo jiné. Navrhujeme analýzu části programu zodpovědné za řešení omezení a její zlepšení.

```
{
"name": "SUT_name",
"t_strength": "2",
"dont_care_values": "no",
"values": "indices",
"parameters": [
  { "identificator": "a", "type": "enum", "blocks":["42", "24", "84", "16",
    "100", "42", "24", "84", "16", "100"] },
  { "identificator": "b", "type": "enum", "blocks":["42", "24", "84", "16",
    "100", "42", "24", "84", "16", "100"] },
  { "identificator": "c", "type": "enum", "blocks":["42", "24", "84", "33"]},
  { "identificator": "d", "type": "enum", "blocks":["42", "24", "84"] },
  { "identificator": "e", "type": "enum", "blocks":["42", "24", "84"] },
  { "identificator": "f", "type": "enum", "blocks":["42", "24"] },
  { "identificator": "g", "type": "enum", "blocks":["42", "24"] },
  { "identificator": "h", "type": "enum", "blocks":["42", "24"] },
  { "identificator": "i", "type": "enum", "blocks":["42", "24"] },
  { "identificator": "j", "type": "enum", "blocks":["42", "24"] },
  { "identificator": "k", "type": "enum", "blocks":["42", "24"] },
  { "identificator": "l", "type": "enum", "blocks":["42", "24"] }
  ],

"constraints": [
  "a.2 -> b.3",
  "c.4 <-> d.1",
  "c.2 or b.1",
  "e.3 and !j.1"
  ]
}
```

Obrázek 5.1: Vstup s omezeními použit pro testování omezení.

Kapitola 6

Závěr

Tato práce se zabývala reimplementací nástroje Combine, který slouží k tvorbě kombinačních testů. Prvním krokem bylo detailní studium metody kombinačního testování a algoritmu IPOG, který je základem pro generování kombinačních testů. Dalším cílem bylo analyzovat nedostatky současné implementace nástroje Combine a navrhnout jejich řešení.

Na základě provedené analýzy byly identifikovány různé slabiny současné implementace, včetně výkonových nedostatků a nespolehlivosti. Jako řešení byla navržena reimplementace nástroje Combine.

Reimplementace nástroje byla provedena s využitím jazyka C++ a zahrnovala implementaci algoritmu IPOG-C, který rozšiřuje původní algoritmus o možnost definování omezení hodnot parametrů. Dále byly použity vhodnější datové struktury pro efektivnější průchod daty a využit multithreading pro zrychlení výpočtů.

Nová verze nástroje, CombineNG, byla následně vyhodnocena a porovnána s původní verzí. Závěrem této práce je úspěšná reimplementace nástroje Combine, která přináší významné vylepšení výkonu. Nová verze nástroje je rychlejší než původní, což přispívá k efektivnějšímu testování softwarových systémů.

Nová verze není bez chyb jak možno vidět v 5, kde jsme identifikovali několik oblastí, které vyžadují pozornost a opravy v budoucím vývoji nástroje Combine. Jsou to refaktorizace lexeru a parseru, oprava chyby v generování řetězce, analýza a zvýšení efektivity výpočtů a oprava delšího počítání sady testů pro vstupy s omezeními. Tyto navrhované úpravy a zlepšení by měly vést k stabilnější a efektivnější verzi nástroje Combine, která bude lépe vyhovovat potřebám uživatelů a bude snadněji udržovatelná v budoucnu.

Literatura

[1] *JSON for Modern C++, v. 3.11.3.* [online], [rev. 2023-11-28], [cit. 2024-03-10]. Dostupné z: `https://github.com/nlohmann/json`.

[2] Červinka, R. *Asistent pro generování testovacích scénářů.* Brno, CZ, 2018. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Aleš Smrčka, Ph. D. Dostupné z: `https://www.vut.cz/studenti/zav-prace/detail/114616`.

[3] Kuhn, D., Kacker, R. a Lei, Y. *Practical Combinatorial Testing.* Special Publication (NIST SP), National Institute of Standards and Technology, Gaithersburg, MD, 2010-10-07 2010. DOI: https://doi.org/10.6028/NIST.SP.800-142.

[4] Lei, Y., Kacker, R., Kuhn, D. R., Okun, V. a Lawrence, J. IPOG: A General Strategy for T-Way Software Testing. In: *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07).* 2007, s. 549–556. DOI: 10.1109/ECBS.2007.47.

[5] Nie, C. a Leung, H. A survey of combinatorial testing. *ACM Comput. Surv.* New York, NY, USA: Association for Computing Machinery. feb 2011, sv. 43, č. 2. DOI: 10.1145/1883612.1883618. ISSN 0360-0300. Dostupné z: `https://doi.org/10.1145/1883612.1883618`.

[6] Yu, L., Lei, Y., Nourozborazjany, M., Kacker, R. N. a Kuhn, D. R. An Efficient Algorithm for Constraint Handling in Combinatorial Test Generation. In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation.* 2013, s. 242–251. DOI: 10.1109/ICST.2013.35.